并发多线程手撕问题总结

1 单体的卖票问题

最好`double check`：

```
public class SellTickets {

    static volatile  int tickets = 100;
    static final Object lock = new Object();

    public static void main(String[] args) throws InterruptedException {
        ExecutorService service = Executors.newFixedThreadPool(6);
        for (int i = 0; i < 6; i++) {
            service.submit(()->{
                while(tickets>0){
                    synchronized (lock){
                        if(tickets>0)
                            System.out.println("No. "+tickets--+" ticket sold by
"+Thread.currentThread().getName());
                    }
                }
            });
        }
        service.shutdown();
    }
}
```

2 两个线程交替打印0-100奇偶数

在问题1的基础上，要求线程间同步：

```
public class ZeroToHundredPrinting {

    static final Object lock = new Object();
    static volatile int count = 0;

    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(2);

        service.submit(()->{
            // int i = 0;
            while(count<=100){
                synchronized (lock){
                    try {
                        lock.wait();
                        if(count>100){
                            continue;
                        }
                        System.out.println("Number "+count+" printed by
"+Thread.currentThread().getName());
                        count++;
                        lock.notify();
                    } catch (InterruptedException e){
                        throw new RuntimeException(e);
                    }
                }
            }
        });

        service.submit(()->{
            // int i = 1;
            try {
                Thread.sleep(100);
                synchronized (lock){
                    lock.notify();
                }
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            while(count<=100){
                synchronized (lock){
                    try {
                        lock.wait();
                        if(count>100){
                            continue;
                        }
                        System.out.println("Number "+count+" printed by
"+Thread.currentThread().getName());
                        count++;
                        lock.notify();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
            }
        });

        service.shutdown();
    }
}
```

这里有一个很有意思的点，一般如果是double check，那么我们把`if(count>100)`放到`synchronized`之后即可，但是！这里我们需要线程同步交替，那么我们必须放到线程苏醒之后，也就是`lock.wait()`之后。

还有一种，是把`notify/notifyAll`放到前面，而`wait`放到后面。

```
public class ZeroToHundredPrinting {

    static final Object lock = new Object();
    static volatile int count = 0;

    public static void main(String[] args) throws InterruptedException {
        ExecutorService service = Executors.newFixedThreadPool(2);
        doSubmit(service);
        Thread.sleep(100);
        doSubmit(service);
        service.shutdown();
    }

    private static void doSubmit(ExecutorService service) {
        service.submit(()->{
            while (count<=100){
                synchronized (lock){
                    System.out.println(Thread.currentThread().getName() + ": " +
count++);
                    lock.notify();
                    if(count<=100){
                        try {
                            lock.wait();
                        } catch (InterruptedException  e){
                            throw new RuntimeException(e);
                        }
                    }
                }
            }
        });
    }
}
```

3 三个线程交替打印0-100

第一种，通过多个条件变量来完成线程间同步：

```
public class ZeroToHundredThreeThread {

    static volatile int count=0;
    static ReentrantLock lock = new ReentrantLock();
    public static void main(String[] args) throws InterruptedException {
        ExecutorService service = Executors.newFixedThreadPool(3);
        Condition cond1 = lock.newCondition();
        Condition cond2 = lock.newCondition();
        Condition cond3 = lock.newCondition();
        doSubmit(service,cond1,cond2);
        Thread.sleep(100);
        doSubmit(service,cond2,cond3);
        Thread.sleep(100);
        doSubmit(service,cond3,cond1);

        Thread.sleep(10000);
        service.shutdown();
    }

    private static void doSubmit(ExecutorService service, Condition waitCond,
Condition notifyCond){
        service.submit(()->{
            while(count<=100){
                try{
                    lock.lock();
                    System.out.println(Thread.currentThread().getName() + ": "+
count++);
                    notifyCond.signal();
                    if(count<=100){
                        try {
                            waitCond.await(20,TimeUnit.MILLISECONDS);
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                    }
                } finally {
                    lock.unlock();
                }
            }
        });
    }
}
```

注意：

- 使用ReentrantLock必须在`finally`块中加上`unlock`；

另一种，自旋：

```
public class ZeroToHundredThreeThread {
    static volatile int count=0;
    static ReentrantLock lock = new ReentrantLock();

    public static void main(String[] args) throws InterruptedException {
        ExecutorService service = Executors.newFixedThreadPool(3);
        doSubmitSpin(service,1);
        Thread.sleep(100);
        doSubmitSpin(service,2);
        Thread.sleep(100);
        doSubmitSpin(service,0);
        Thread.sleep(10000);
        service.shutdown();
    }

    private static void doSubmitSpin(ExecutorService service, int spinValue){
        service.submit(()->{
            while(count<=100){
                if(count % 3==spinValue){
                    try {
                        lock.lock();
                        if(count % 3 != spinValue || count >100 ){
                            continue;
                        }
                        System.out.println(Thread.currentThread().getName() + ":
"+ count++);
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    } finally {
                        lock.unlock();
                    }
                }
            }
        });
    }
}
```

这里考虑一下continue和finally的关系，事实上finally会在continue、return、break指令前执行，但仍然不推荐使用跳转语句，这可能会导致错误无法正常抛出。

```
private static void doSubmitSpin(ExecutorService service, int spinValue){
    service.submit(()->{
        while(count<=100){
            if(count % 3==spinValue){
                try {
                    lock.lock();
                    if(count % 3 == spinValue && count<=100 ){
                        System.out.println(Thread.currentThread().getName() + ":
"+ count++);
```

```
                    }
                } finally {
                    lock.unlock();
                }
            }
        }
    });
}
```

4 同时转账问题

```
public class TransferConcurrently {

    static class Account{
        private int balance;

        public Account(int balance){
            this.balance = balance;
        }

        @Override
        public String toString() {
            return "Account{" +
                    "balance=" + balance +
                    '}';
        }

        @Override
        public int hashCode(){
            return this.balance;
        }
    }


    static final Object extraLock = new Object();
    public static void main(String[] args) {
        Account account1 = new Account(300);
        Account account2 = new Account(500);
        ExecutorService service = Executors.newFixedThreadPool(2);
        // doTransfer(service,account1,account2,100);
        // doTransfer(service,account2,account1,50);
doTransferGetLockByHash(service,account1,account2,100);
        doTransferGetLockByHash(service,account2,account1,50);

        Account account3 = new Account(500);
        Account account4 = new Account(500);
        doTransferGetLockByHash(service,account3,account4,100);
        doTransferGetLockByHash(service,account4,account3,300);
        service.shutdown();
    }


    private static void doTransfer(ExecutorService service, Account from, Account
to, int money){
        service.submit(()->{
            transferLockInOrder(from, to, money);
        });
    }

    private static void doTransferGetLockByHash(ExecutorService service, Account
from, Account to, int money){
        service.submit(()->{
            if(from.hashCode()<to.hashCode()){
                transferLockInOrder(from,to,money);
            } else {
                transferLockInReverse(from,to,money);
            }
        });
    }

    private static void transferSolveSameHashCode(ExecutorService service,Account
from ,Account to,int money){
        service.submit(()->{
            if(from.hashCode()==to.hashCode()){
                synchronized (extraLock){
                    transferLockInOrder(from, to, money);
                }
            }
            else {
                if(from.hashCode()<to.hashCode()){
                    transferLockInOrder(from,to,money);
                } else {
                    transferLockInReverse(from,to,money);
                }
            }
        });
    }

    private static void transferLockInOrder(Account from, Account to, int money)
{
        synchronized (from){
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            System.out.println(Thread.currentThread().getName() + " got lock " +
from);
            synchronized (to){
                System.out.println(Thread.currentThread().getName() + " got lock
" + to);
                from.balance = from.balance-money;
                to.balance = to.balance + money;
                System.out.println(Thread.currentThread().getName() + " transfer
done " + from +" and "+to);
            }
        }
    }


    private static void transferLockInReverse(Account from, Account to, int
money){
        synchronized (to){
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            System.out.println(Thread.currentThread().getName() + " got lock " +
from);
            synchronized (from){
                System.out.println(Thread.currentThread().getName() + " got lock
" + to);
                from.balance = from.balance-money;
                to.balance = to.balance + money;
                System.out.println(Thread.currentThread().getName() + " transfer
done " + from +" and "+to);
            }
        }
    }
}
```

代码很多，其实就两个点要注意：

- 通过相同加锁顺序来防止死锁，但交易账户没有天然顺序，所以我们用hashcode解决，但hashcode是可能碰撞的；
- 一旦hashcode碰撞，我们就需要引入额外的锁来完成转账操作，相当于升级了。

5 哲学家进餐问题

解决策略：

- 服务员检查（避免策略）：由服务员进行判断分配，如果发现可能会发生死锁，不允许就餐；
- 改变一个哲学家拿叉子的顺序（避免策略）：改变其中一个拿的顺序，破坏环路；

- 餐票（避免策略）：吃饭必须拿餐票，餐票一共只有4张，吃完了回收；
- 领导调节（检测与恢复策略）：定时检查，如果发生死锁，随机剥夺一个的筷子。

展示问题：

```
class Philosopher implements Runnable {

    private Object leftChopstick;
    private Object rightChopstick;

    public Philosopher(Object leftChopstick, Object rightChopstick) {
        this.leftChopstick = leftChopstick;
        this.rightChopstick = rightChopstick;
    }

    public static void main(String[] args) {
        //定义5个哲学家
        Philosopher[] philosophers = new Philosopher[5];
        //定义筷子
        Object[] chopticks = new Object[philosophers.length];
        //初始化筷子
        for (int i = 0; i < chopticks.length; i++) {
            chopticks[i] = new Object();
        }
        for (int i = 0; i < philosophers.length; i++) {
            Object leftChopstick = chopticks[i % philosophers.length];
            Object rightChopstick = chopticks[(i + 1) % philosophers.length];
            philosophers[i] = new Philosopher(leftChopstick, rightChopstick);
            new Thread(philosophers[i], "哲学家" + (i + 1)).start();
        }

    }

    @Override
    public void run() {
        try {
            while (true) {
                doAction("think");
                synchronized (leftChopstick) {
                    doAction("拿起左手边筷子");
                    synchronized (rightChopstick) {
                        doAction("拿起右手边筷子");
                        doAction("放下右手边筷子");
                    }
                    doAction("放下左手边筷子");
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    //打印在做的事情，并随机睡眠一段时间
    static void doAction(String action) throws InterruptedException {
        System.out.println(Thread.currentThread().getName() + ":" + action);
        Thread.sleep((long)Math.random()*1000);
    }
}
```

改变一个哲学家拿起的顺序：

```
  for (int i = 0; i < philosophers.length; i++) {
            Object leftChopstick = chopticks[i % philosophers.length];
            Object rightChopstick = chopticks[(i + 1) % philosophers.length];
            if (i == philosophers.length - 1) {
                philosophers[i] = new Philosopher(rightChopstick, leftChopstick);
            } else {
                philosophers[i] = new Philosopher(leftChopstick, rightChopstick);
            }
            new Thread(philosophers[i], "哲学家" + (i + 1)).start();
        }
```

6 赛马

淘天真题

10匹马， 10000m跑道，裁判发号口令，10匹马需要同时进入赛道， 当10匹马全部冲过终点， 裁判宣布成绩（打印10匹马的排名）

CountDownLatch。